PARSY: Performance Aware Reconfiguration of Software Systems

MORENO MARZOLLA\textsuperscript{1}, and RAFFAELA MIRANDOLA\textsuperscript{2}

\textsuperscript{1}Università di Bologna, Bologna, Italy
\textsuperscript{2}Politecnico di Milano, Milano, Italy

(Received on December 12, 2010 and revised on March 09, 2011)

Abstract: Dynamic reconfiguration of component-based systems is recognized as a viable way to meet quality requirements in a mutable operating environment. In this paper we consider the problem of maintaining the overall response time of a component-based system below a given threshold, as the system is subject to variable workload. We assume that the system components can be dynamically reconfigured to provide a degraded service with lower response time. Each component operating at one of the available quality levels is assigned a utility, with higher quality levels associated to higher utilities. The main contribution of this paper is an on-line algorithm for performance-aware reconfiguration of degradable software systems called PARSY (Performance Aware Reconfiguration of software SYstems). PARSY tunes individual components in order to maximize the system utility with the constraint of keeping the system response time below a specific threshold. PARSY drives the runtime dynamic reconfiguration step with the help of a Queuing Network performance model. Numerical experiments are used to illustrate the effectiveness of the proposed approach.

Keywords: Dynamic reconfiguration, software performance modeling, scalability

1. Introduction

The component-based software development paradigm allows complex systems to be built by assembling a number of independent components, each one providing a specific functionality. This paradigm has many advantages, including the ability of reducing development costs by allowing components reuse.

One crucial problem is building the system such that non-functional requirements, like reliability or performance, can be satisfied as well. This is difficult for several reasons. First, it is not easy to infer properties of the whole system by considering its components in isolation. Furthermore, performance depends on the external workload: even if the system has been implemented to sustain the expected workload, there may be unexpected variabilities that were not accounted for, and cause sporadic slowdowns. Thus, in order to ensure that performance related non-functional requirements are satisfied, both proper capacity planning and some form of adaptation are necessary.

In this paper, we focus on the system response time as a specific performance metric. The system response time is important because it impacts directly on users of the system. In general, the response time increases with the number of users concurrently accessing the system, unless the system adopts appropriate strategies to reconfigure itself in order to cope with spikes in the workload.

One commonly used solution is based on dynamic scalability using more physical resources. For example, an E-commerce Web site might react to an increased load by
deploying the front-end tier across more Web servers and load-balancing the requests among the available servers [1]. However, there are situations where this approach is not practical: not every system can scale by simply adding more resources; furthermore, it could be difficult to react quickly to spikes in the workload, as allocating new resources and starting new application instances is not instantaneous.

In this paper we propose an approach, called Performance Aware Reconfiguration of software SYstems (PARSY), whose goal is to reduce the system response time through a performance-aware degradation of the application, driven by the solution of performance models at runtime. Specifically, we consider a component-based software system, where some of the components can provide a degraded (but still acceptable) service with reduced response time (see Figure 1). Going back to the example of E-commerce site under heavy load, instead of relying on more server instances, the system might provide a degraded service for some non-critical components. For example, the system could show heavily compressed product images in order to cut processing and data transfer times. The decision, of which components can be degraded, while still providing an acceptable service quality, is in general application-dependent.

We assume that the system administrators define a maximum allowed system response time. The system is enhanced with a monitoring component that identifies violations of the response time constraint. When violations occur, PARSY uses a Queuing Network (QN) performance model to decide which component(s) of the system should be degraded. As the workload fluctuates, PARSY is able to degrade or upgrade individual components in order to satisfy, if possible, the response time constraint. The system administrators can associate weights (utility values) to each configuration, such that PARSY can choose to degrade less important components first.

This paper is structured as follows: in Section 2 we define the system model and the optimization problem we are addressing. In Section 3 we describe how a solution to the optimization problem can be efficiently computed. In Section 4 we evaluate the solution algorithm on some test instances, while a short survey of the related works is presented in Section 5. Finally, conclusions and future improvements are discussed in Section 6.

![Figure 1: PARSY System Model Assumptions](image)

2. System Model

In PARSY we consider a software system made of $K$ components $C_1, ..., C_K$. Each component $C_i$ can be configured to provide service at different quality levels $L_1, ..., L_L$. The idea is that each component can be degraded to provide sub-optimal service faster. The system exhibits a tradeoff between fast but less accurate computations, and slow but accurate computations.
In the following we assume that each component $C_k$ performs a single operation. We further assume that all components are independent, in the sense that each one is hosted on a different physical resource. Note that both these limitations can be addressed using a more powerful performance model based on multiclass Queuing Networks. A component implementing multiple operations can be modeled as a queuing center serving multiple job classes, each class corresponding to an operation. In the same way it would be possible to model multiple components sharing the same underlying resource (e.g., multiple servers running on the same processor). Multiclass QNs can model the interference caused by shared resource as well as internal interactions of multiple operations on the same component. The drawback is that multiclass QNs are computationally expensive to analyze [6], and thus not suitable to be used for run-time adaptation as required by PARSY.

A system configuration is a vector $l = (l_1, ..., l_K)$ such that for each $k = 1, ..., K$, $l_k \in \{1, ..., L_k\}$; this indicates that component $C_k$ is operating at quality level $l_k$. We denote with $D(k,j)$ the mean service demand of component $C_k$ when operating at level $j \in \{1, ..., L_k\}$. Quality levels directly impact the execution time of components: a component $C_k$ running at level $j > i$ must require less resources than the same component running at higher quality level $i$: $D(k,i) < D(k,j)$. From this we have that service demands must be monotonically increasing: $0 < D(k,1) < D(k,2) < ... < D(k,L_k)$. When a component is degraded, it operates faster, but only produces an “approximate” result (if some component $C_k$ can not be degraded, we let $L_k = 1$). Obviously, the system administrator is more interested in keeping components at the highest possible quality levels; degraded components are acceptable only under heavy load.

In PARSY, system administrators define a positive utility value $UT(k,j)$ for component $C_k$ operating at quality level $j$. The utility value allows system administrators to define weights associated with levels, such that unimportant services are considered for degradation before important ones. For each $C_k$ we require that $0 < UT(k,1) < UT(k,2) < ... < UT(k,L_k)$.

The utility of system configuration $l$ is defined as the sum of utilities of each component:

$$ UT(l) = \sum_{k=1}^{K} UT(k,l_k) $$

(1)

The initial system configuration is $(L_1, ..., L_K)$, such that all components operate at the best quality level.

The goal of PARSY is to selectively upgrade or degrade some of the components such that the estimated system response time is kept below a predefined threshold $R_{max}$, and the system utility is the highest possible. In other words, we seek a solution to the following optimization problem:

$$ \text{maximize} \quad UT(l) $$

subject to \quad $R(N,l) < R_{max}$ \quad (2)

$$ l_k \in \{1, ..., L_k\} \quad k = 1, ..., K $$

where $R(N,l)$ is the estimated system response time with configuration $l$, when there are $N$ concurrent users accessing the system. We are seeking the configuration that maximizes the total utility and keeps the estimated response time below the threshold $R_{max}$. 
Finding the optimal solution to (2) by evaluating all possible system configurations would require time $O(f(N,K) \prod_{k=1}^{K} L_k)$, where $f(N, K)$ is the cost of estimating $R(N;l)$. Note that the special nature of the constraint makes the optimization problem (2) different from, e.g., the Knapsack Problem [2]; thus, specific solution techniques must be developed. Also, observe that the problem (2) could even have no solution at all. In fact, the lowest possible response time can be achieved when the system is in its configuration $(1, ..., 1)$. In this state, performance can no longer be improved by degrading components, as no component can be further degraded.

**Table 1:** Symbols used in this paper

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C_1, ..., C_K$</td>
<td>Components</td>
</tr>
<tr>
<td>$R_{\text{max}}$</td>
<td>Maximum allowed system response time</td>
</tr>
<tr>
<td>$X$</td>
<td>Measured system throughput</td>
</tr>
<tr>
<td>$R$</td>
<td>Measured system response time</td>
</tr>
<tr>
<td>$N$</td>
<td>Number of concurrent users (computed according to Eq. (4))</td>
</tr>
<tr>
<td>$l=(l_1, ..., l_K)$</td>
<td>System configuration (component $C_k$ is operating at level $l_k$)</td>
</tr>
<tr>
<td>$L_k$</td>
<td>Number of quality levels offered by component $C_k$</td>
</tr>
<tr>
<td>$R(N;l)$</td>
<td>Estimated system response time at configuration $l$ with $N$ requests</td>
</tr>
<tr>
<td>$D(k,j)$</td>
<td>Average service demand of component $C_k$ at quality level $j$</td>
</tr>
<tr>
<td>$U(k,j)$</td>
<td>Utility of component $C_k$ when operating at quality level $j$</td>
</tr>
</tbody>
</table>

A basic result from queuing theory states that the response time $R(N;l)$ is asymptotically bound by [3]:

$$R(N;l) \geq R(N;1, ..., 1) \geq N \max_k \{D(k,1)\}$$

Thus, from (3) we conclude that, for any configuration $l=(l_1, ..., l_K)$, the response time $R(N;l)$ goes to infinity as the number of requests $N$ increases, which means that for sufficiently high loads the constraint $R(N;l) < R_{\text{max}}$ can not be satisfied. In this case, PARSY will select the configuration $(1, ..., 1)$ as solution of (2).

Table 1 summarizes the symbols used in this paper.

### 3. Model Solution

We now propose a practical way to find an approximate solution to the problem (2) above. The main points of PARSY are the following (see Figure 2a): (i) we enhance the system with a monitoring component which triggers a reconfiguration whenever the measured system response time deviates from the threshold $R_{\text{max}}$ (see Section 3.1); and (ii) we use a closed QN model to estimate response time $R(N;l)$ for a given system configuration $l$ (see Section 3.2).

#### 3.1 Identifying Reconfiguration Times

An important problem we need to address it to decide when a reconfiguration should occur. To do so, we enhance the software system with a monitoring component. The monitor is a passive observer that measures the system response time $R'$ and throughput $X'$. The monitor notifies a separate component (called controller) when a reconfiguration should take place.
If the measured response time is less than the threshold \( R' < R_{\text{max}} \), the monitor notifies the controller to upgrade components, if possible; if the measured response time is greater than the threshold \( R' > R_{\text{max}} \), the monitor notifies the controller to downgrade components.

Attention must be paid to avoid unnecessary reconfigurations when the measured response time \( R' \) bounces above and below the threshold \( R_{\text{max}} \). A common approach to deal with this situation is to trigger a reconfiguration after the event \( R' > R_{\text{max}} \) (resp. \( R' < R_{\text{max}} \)) has been observed multiple consecutive times. Alternatively, we can define two thresholds \( R_{\text{min}} \) and \( R_{\text{max}} \) such that an upgrade is triggered when \( R' < R_{\text{min}} \), and a downgrade is triggered when \( R' > R_{\text{max}} \). It is also important to wait for the system to settle down after a reconfiguration. For a recent result see [4].

Figure 2a: System Architecture, including the Monitor and the Controller.

Figure 2b: Queuing Network Performance Model

3.2 Finding a New Configuration

The controller is responsible for identifying a configuration \( l \) that solves the optimization problem (2). The controller uses a simple greedy heuristic in which individual components are upgraded (or downgraded). At each step the system response time \( R(N;l) \) is estimated using a single-class, closed QN model.

The Queuing Network model contains \( K \) service centers, where center \( k \) corresponds to component \( C_k \) (Figure 2b). If the system configuration is \( l = (l_1, \ldots, l_K) \), then the service demand of queuing center \( k \) is \( D(k, l_k) \). We assume that the QN has product-form solution; this assumption puts some constraints on the performance model (see [3, 6] for details), which could reduce its accuracy. However, it should be observed that building a more accurate performance model requires detailed knowledge of the real system, which may not be readily available; furthermore, detailed models can be very difficult to analyze, which is a major concern in situations where model-based, run-time adaptation is needed.
We also observe that the use of a simple QN model makes PARSY system-independent and thus very general.

To analyze the closed QN model, we need the number \( N \) of requests in the system at the time of reconfiguration. \( N \) can be computed from the observed values \( X' \) and \( R' \) using Little’s law [5]:

\[
N = X' R'
\]

At this point, we can estimate \( R(N; l) \) using Mean Value Analysis (MVA) [6] (Algorithm 3). The computational complexity of MVA is \( O(NK) \), where \( N \) is the number of active requests and \( K \) is the number of service centers (components in the system). A faster way to estimate the response time is to compute asymptotic upper and lower bounds on the response time of QN model (Algorithm 4). Response time bounds can be computed in time \( O(K) \) [3], but only give approximations on the true response time of the QN.

We now describe a greedy algorithm that identifies a feasible solution to (2) as follows:

- If \( R' > R_{\text{max}} \), we keep degrading components until the estimated system response time, as computed using the QN model, becomes less than the threshold \( R_{\text{max}} \).
- If \( R' < R_{\text{max}} \), we keep upgrading components as long as the estimated system response time remains less than the threshold \( R_{\text{max}} \).

Let us analyze the two cases in detail.

**Degrading Components:** When \( R' > R_{\text{max}} \), a reconfiguration is triggered by executing Algorithm 1. At each iteration, the algorithm degrades the component \( C_B \) for which the ratio \( D(B, l_B) / UT(B, l_B) \) is maximum, where \( l_B \) is the quality level of \( C_B \) after it has been degraded. The algorithm stops when either (i) the estimated response time is below the threshold, or (ii) all components have been degraded at quality level 1, so that no further degradation is possible. The idea is to degrade the component with both a high service demand and a low utility. In queuing theory, the center with maximum demand is the bottleneck device; here we also take into account the utility of the degraded component.

**Upgrading Components:** Algorithm 2 is used to upgrade components as long as the estimated system response time \( R(N; l) \) remains below the threshold \( R_{\text{max}} \). Again, we use a greedy approach in which a component \( C_U \) to be upgraded is chosen at each iteration. Let \( l_U \) be the quality level of component \( C_U \) before the reconfiguration. Then, \( C_U \) is chosen to satisfy the following two properties:

- After upgrading \( C_U \) at configuration \((l_U + 1)\), the new estimated system response time is below the threshold \( R_{\text{max}} \).
- \( C_U \) is the component whose upgrade provides the maximum utility with the minimum increase in system response time.

This approach is similar to the greedy algorithm for solving the knapsack problem [2], where items are tried in order of decreasing unitary value.

### 3.3 Computational Complexity and Scalability Considerations

Both Algorithms 1 and 2 execute a number of iterations in which a single component is degraded/upgraded; in particular, at each iteration one component, say \( C_k \), can be upgraded from level \( l_k \) to level \((l_k + 1)\), or degraded from level \( l_k \) to level \((l_k - 1)\). The worst case for Algorithms 1 and 2 happens when the whole system is degraded from configuration \((L_1, \ldots, L_K)\) to \((1, \ldots, 1)\), or the other way. Thus, in the worst case at most \( \sum_{k=1}^{K} L_k \) iterations are performed. The cost of each iteration is dominated by the cost \( f(N,K) \) of evaluating \( R(N; l) \). If the system response time is estimated using MVA [6], we have that \( f(N,K) = O(NK) \) which implies that the computational complexity of
Algorithms 1 and 2 is $O(NK \sum_{k=1}^{K} L_k)$. If the system response time is estimated by computing upper and lower Balanced System Bounds [3] $R^+$ and $R^-$ and letting $R(N; l) = (R^+ + R^-)/2$, then we have $f(N, K) = O(K)$. In this case we can reduce the computational complexity of Algorithms 1 and 2 to $O(K \sum_{k=1}^{K} L_k)$, which is independent from the number of requests $N$. The MVA algorithm computes the exact value of the response time, at the cost of a higher computational complexity; on the other hand, performance bounds provide a quick estimate of the system response time. It is important to remember that PARSY operates at run-time: new configurations must be computed quickly so that the system has a chance to promptly react to workload burstiness. Furthermore, an accurate solution of the QN model is really not important, as the model is just a rough approximation of the actual system. These considerations suggest that Bounds analysis is likely to be more appropriate. This is also confirmed by the experimental results shown in the next section: PARSY+MVA identifies system configurations which are only marginally better than those identified by PARSY+BSB.

4. Numerical Example

In this section we assess the effectiveness of PARSY by means of a set of synthetic test cases. We consider a software system with $K$ components such that each component can operate at $L$ different quality levels. We experiment with multiple combinations of $K$ and $L$: we use $K = 10, 20, 30, 50$ and $L = 2, 3, 5$. For all $k = 1, ..., K$, $j = 1, ..., L$, the service demands $D(k, j)$ and utilities $U_T(k, j)$ of component $C_k$ operating at quality level $j$ are randomly generated when each model is created.

We performed a time-stepped simulation in which we evaluate each system for $T = 200$ time steps. The number of users (requests) $N_t$ at step $t = 1, ..., T$ is produced using a random walk model. For each experiment the threshold $R_{max}$ is defined as:

$$R_{max} = \max \{ R(N_t; 1, ..., 1) | t = 1, ..., T \}$$

that is, $R_{max}$ is the maximum system response time, over all values of $N_t$, when all components operate at quality level 1 (worst). This ensures that there always exists a configuration such that the system response time is kept below $R_{max}$ for all values of $N_t$. At least, configuration $(1, ..., 1)$ always satisfies the constraint in the optimization problem (2).

We implemented Algorithms 1 and 2 in GNU Octave [7]. Initially, all components are set at the best quality level. We tested two different techniques to estimate $R(N; l)$: (i) using MVA and (ii) using Balanced System Bounds (BSB).

PARSY is executed on-line, that is it finds a new configuration at time step $t$ by considering only the configuration at the previous step $t-1$ and the number of currently active requests $N_t$. The “observed” system response time $R'(t)$ at time $t$ is computed using MVA on the QN system model. The value of $R'(t)$ is then used to decide whether the configuration should be upgraded or downgraded, as described in Section 3. For each configuration we also compute the utility according to (1).
Figure 3 shows an example of the results for a system with $K = 10$ components operating at $L = 2$ quality levels. The top part of the plot shows the observed system response time for the static configuration $(L, \ldots, L)$, PARSY+MVA and PARSY+BSB respectively. Reconfiguration points are also shown. The middle part of Figure 3 shows the utility over time for the static system, as well as using PARSY. Note that the system utility with configuration $(L, \ldots, L)$ is, by construction, an upper bound of the utility of any other valid configuration. Finally, the bottom part of Figure 3 shows the number of users $N_t$ at time step $t$.

In order to compare PARSY+MVA and PARSY+BSB we consider two metrics: the total utility $U_T$ and the response time overflow $\Delta R$. The total utility is the sum of the utility achieved at each simulation step; this is the area under the curves in the central plot of Figure 3. The response time overflow is the area which lies above the line $y = R_{\text{max}}$ and below $y = R(t)$. The total utility $U_T$ is a “higher is better” metric, while the response time overflow $\Delta R$ is a “lower is better” metric.

We report in Table 2 the results of all experiments; for a better visual comparison, the same results are shown in Figure 4. We observe that PARSY is very effective in reducing the response time overflow; at the same time the total utility is kept as a fraction of the maximum possible value. It is interesting to observe that, in the considered test cases, PARSY+BSB produces only marginally worse configurations than those produced by PARSY+MVA. This means that PARSY+BSB achieve a slightly lower response time overflow but with lower utility than the configurations identified by PARSY+MVA.
Table 2: Results of all the Experiment Sets: K is the number of components; L is the number of quality levels; UT is the total utility and 𝛿R the response time overflow.

<table>
<thead>
<tr>
<th>K</th>
<th>L</th>
<th>UT</th>
<th>AR</th>
<th>UT</th>
<th>AR</th>
<th>UT</th>
<th>AR</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>2</td>
<td>6033.61</td>
<td>130988.64</td>
<td>3487.51</td>
<td>760.10</td>
<td>3448.06</td>
<td>612.70</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>13770.77</td>
<td>59477.44</td>
<td>11457.83</td>
<td>289.72</td>
<td>11329.68</td>
<td>289.72</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>23599.01</td>
<td>438844.37</td>
<td>11975.59</td>
<td>2598.19</td>
<td>11748.92</td>
<td>2597.51</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
<td>20521.15</td>
<td>94963.73</td>
<td>16658.64</td>
<td>413.75</td>
<td>16348.54</td>
<td>356.21</td>
</tr>
<tr>
<td>10</td>
<td>3</td>
<td>34280.73</td>
<td>230470.96</td>
<td>20815.32</td>
<td>1618.33</td>
<td>20338.00</td>
<td>1345.80</td>
</tr>
<tr>
<td>10</td>
<td>5</td>
<td>55286.46</td>
<td>395327.45</td>
<td>32003.47</td>
<td>2645.46</td>
<td>31360.16</td>
<td>2235.00</td>
</tr>
<tr>
<td>20</td>
<td>2</td>
<td>38317.48</td>
<td>89193.94</td>
<td>33866.14</td>
<td>740.43</td>
<td>32994.29</td>
<td>505.57</td>
</tr>
<tr>
<td>20</td>
<td>3</td>
<td>62055.95</td>
<td>240330.28</td>
<td>42062.21</td>
<td>2217.03</td>
<td>40712.85</td>
<td>1222.02</td>
</tr>
<tr>
<td>20</td>
<td>5</td>
<td>101218.02</td>
<td>385216.80</td>
<td>50825.48</td>
<td>3238.69</td>
<td>49605.56</td>
<td>2419.09</td>
</tr>
<tr>
<td>50</td>
<td>2</td>
<td>95497.09</td>
<td>112307.36</td>
<td>80063.05</td>
<td>1376.63</td>
<td>74014.90</td>
<td>684.58</td>
</tr>
<tr>
<td>50</td>
<td>3</td>
<td>141301.04</td>
<td>236503.27</td>
<td>95328.14</td>
<td>2506.65</td>
<td>89677.31</td>
<td>1547.18</td>
</tr>
<tr>
<td>50</td>
<td>5</td>
<td>244140.53</td>
<td>506933.84</td>
<td>98342.00</td>
<td>3999.99</td>
<td>94645.23</td>
<td>2844.83</td>
</tr>
</tbody>
</table>

The difference in the utility value is quite small, and is compensated by the fact that BSB are faster to compute, and thus are better suited for large systems with many components and a large number of concurrent requests. Note that efficiency of the reconfiguration algorithms is hardly an issue for small to medium size system. Our tests have been performed on a Linux PC (kernel 2.6.24) with an AMD Athlon 64 X2 Dual Core processor 3800+ with 2GB of RAM, using GNU Octave version 3.2.3. On this system, a single reconfiguration step requires negligible time using MVA even for the largest test case with K = 50 components and L = 5 levels.

Figure 4: Utility and Response Time Overflow for the Experiments (Labels on the horizontal axes denote the parameters K and L used for that experiment. The response time overflow is shown in log scale)
5. Related Works

In the last few years, as outlined in [8], the topic of reconfigurable and self-adaptive computing systems has been studied in several communities and from different perspectives. The autonomic computing framework is a notable example of general approach to the design of such systems [9]. Hereafter, we focus on works appeared in the literature dealing with the self-adaptation of software systems to guarantee the fulfillment of QoS requirements. Specifically, we focus on works that make use of models to perform this step. GPAC (General-Purpose Autonomic Computing), for example, is a tool-supported methodology for the model-driven development of self-managing IT systems [10]. The core component of GPAC is a generic autonomic manager capable of augmenting existing IT systems with a MAPE autonomic computing loop. The GPAC tools and the probabilistic model checker PRISM [11] are used together successfully to develop autonomic systems involving dynamic power management and adaptive allocation of data-center resources [12]. KAMI [13] is another framework for model evolution by runtime parameter adaptation. KAMI focuses on Discrete Time Markov Chain models that are used to reason about non-functional properties of the system. The authors adapt the QoS properties of the model using Bayesian estimations based on runtime information, and the updated model allows the verification of QoS requirements. The Models@Run.Time approach [14] proposes to leverage software models and to extend the applicability of model-driven engineering techniques to the runtime environment to enhance systems with dynamic adapting capabilities. In [15], the authors use an architecture-based approach to support dynamic adaptation. In [16] the authors describe a methodology for estimation of model parameters through Kalman filtering. This work is based on a continuous monitoring that provides run-time data feeding a Kalman filter, aimed at updating the performance model.

In the area of service-based systems (SBS), devising QoS-driven adaptation methodologies is of utmost importance in the envisaged dynamic environment in which they operate. Most of the proposed methodologies for QoS-driven adaptation of SBS address this problem as a service selection problem (e.g., [17,18]). Other papers have instead considered service-based adaptation through workflow restructuring, exploiting the inherent redundancy of SBS (e.g., [19,20]). In [21] a unified framework is proposed where service selection is integrated with other kinds of workflow restructuring, to achieve a greater flexibility in the adaptation.

The works closest to our approach are [22–24]. In [22], the authors propose a conceptual model dealing with changes in dynamic software evolution. Besides, they apply this model to a simple case study, in order to evaluate the effectiveness of fine-grained adaptation changes like service-level degrading/upgrading action considering also the possibility to perform actions involving the overall resource management. The approach proposed in [23] deals with QoS-based reconfigurations at design time. The authors propose a method based on evolutionary algorithms where different design alternatives are automatically generated and evaluated for different quality attributes. In this way, the software architect is provided with a decision making tool enabling the selection of the design alternatives that best fits multiple quality objectives. Menascè et al. [24] have developed the SASSY framework for generating service-oriented architectures based on quality requirements. Based on an initial model of the required service types and their
communication, SASSY generates an optimal architecture by selecting the best services and potentially adding patterns such as replication or load balancing. With respect to existing approaches, PARSY lies in the research line fostering the usage of models at runtime to drive the QoS-based system adaptation. The proposed approach uses two very efficient modeling and analysis techniques that can then be used at runtime without undermining the system behavior and its overall performance.

6. Conclusions

In this paper we presented PARSY, a framework for runtime performance aware reconfiguration of component-based software systems. The idea underlying PARSY is to selectively degrade and upgrade system components to guarantee that the overall response time does not exceed a predefined threshold. The capability of driving this dynamic degradation is achieved through the introduction of a monitoring component that triggers a reconfiguration whenever the measured response time exceeds the threshold, and the use of a QN model to estimate, at run-time, the response time of various reconfiguration scenarios. The response times are used to feed the optimization model whose solution gives the system configuration that maximizes the total utility while keeping the response time below the threshold.

The methodology proposed in this paper can be improved along several directions. It would be useful to explore the use of forecasting techniques as a mean to trigger system reconfiguration in a proactive way. Another direction that deserves further investigation is the use of different numerical techniques for an efficient and accurate solution of the optimization problem.

References


Moreno Marzolla graduated in Computer Science from "Ca' Foscari" University of Venezia (Italy) in 1998, and received a Ph.D. in Computer Science from the same University in 2004. From 1998 to 2001 he worked at the Italian National Institute for Nuclear Physics (INFN) as a software developer. From 2004 to 2005 he was a post-doc researcher at the "Ca' Foscari" University of Venezia. From 2005 to 2009 he was again with INFN as a Software Engineer, working in the area of Grid Computing supported by the EGEE, OMII-Europe and EGEE-3 EU-funded projects. In November 2009 he joined the Department of Computer Science of the University of Bologna, where he is currently assistant professor. His research interests include performance modeling of complex systems and high performance and Cloud computing.

Raffaela Mirandola is an Assistant Professor at Dipartimento di Elettronica e Informazione at Politecnico di Milano. Raffaela's research interests are in the areas of performance and reliability modeling and analysis of software/hardware systems with special emphasis on the automatic generation of performance models for component-based systems and service-based systems, model driven engineering applied to embedded systems, empirical software engineering and software process analysis. She has published over 70 journal and conference articles on these topics. She served and is currently serving in the program committees of conferences in the research areas and she is a member of the Editorial Board of Journal of System and Software (published by Elsevier). She has been involved in several national and European research projects among which EU project PLASTIC, CASCADAS, Q-ImPreSS (FP7-215013) and IDEAS SMScom.
Algorithm 1: Degrade configuration

Input: \( l \) current system configuration
Input: \( N \) number of requests in the system
Input: \( D(k, j) \) service demand of component \( C_k \) operating at quality level \( j \)
Input: \( UT(k, j) \) utility of component \( C_k \) operating at quality level \( j \)

Output: \( l_{\text{new}} \) is the new system configuration

\[
l_{\text{new}} := \{ k \mid l_{\text{new}}(k) > 1 \}
\]

[Candidate set of components which can be degraded]

while \( (C \neq \emptyset) \) do

\[
B := \text{arg max}_{k \in C} \left( \frac{D(k, l_{\text{new}}(k))}{UT(k, l_{\text{new}}(k))} \right)
\]

\[
l_{\text{new}}(B) := l_{\text{new}}(B) - 1
\]

[Degrade \( C_B \)]

Estimate \( R(N; l_{\text{new}}) \) using the MVA or BSB algorithms

if \( (R(N; l_{\text{new}}) < R_{\text{max}}) \) then

Break

else

\[
C := \{ k \mid l_{\text{new}}(k) > 1 \}
\]

[Recompute the candidate set]

end

Return \( l_{\text{new}} \)

Algorithm 2: Upgrade configuration

Input: \( l \) current system configuration
Input: \( N \) number of requests in the system
Input: \( D(k, j) \) service demand of component \( C_k \) operating at quality level \( j \)
Input: \( UT(k, j) \) utility of component \( C_k \) operating at quality level \( j \)

Output: \( l_{\text{new}} \) is the new system configuration

\[
l_{\text{new}} := \{ k \mid l_{\text{new}}(k) > 1 \}
\]

[Candidate set of components which can be upgraded]

while \( (C \neq \emptyset) \) do

\[
U := \text{arg min}_{k \in C} \left( \frac{D(k, l_{\text{new}}(k))}{UT(k, l_{\text{new}}(k))} \right)
\]

\[
l_{\text{new}}(U) := l_{\text{new}}(U) + 1
\]

[Try to upgrade \( C_U \)]

Estimate \( R(N; l_{\text{new}}) \) using the MVA or BSB algorithms

if \( (R(N; l_{\text{new}}) > R_{\text{max}}) \) then

\[
l_{\text{new}}(U) := l_{\text{new}}(U) - 1
\]

[Rollback configuration for \( C_U \)]

else

\[
C := C \setminus U
\]

[Recompute the candidate set]

end

Return \( l_{\text{new}} \)

Algorithm 3: Response time estimation using Mean Value Analysis

Input: \( l \) current system configuration
Input: \( N \) number of requests in the system
Input: \( D(k, j) \) service demand of component \( C_k \) operating at quality level \( j \)

Output: \( R \) estimated system response time

for all \( k = 1, 2, \ldots, K \) do

\[
Q_k := 0
\]

end

for all \( n = 1, 2, \ldots, N \) do

for all \( k = 1, 2, \ldots, K \) do

\[
R_k := D(k, l_k) \times (1 + Q_k)
\]

[Residence time at \( C_k \)]

end

\[
X := \frac{nR}{R}
\]

[System response time]

end

for all \( k = 1, 2, \ldots, K \) do

end

System throughput]
Algorithm 4: Response time estimation using Balanced System Bounds

**Input:**
- \( l \): current system configuration
- \( N \): number of requests in the system
- \( D(k,j) \): service demand of component \( C_k \) operating at quality level \( j \)

**Output:** \( R \): estimated system response time

\[ D_{\text{max}} := \max \{ D(k,h_i) \mid k := 1, \ldots, K \} \]
\[ D_{\text{tot}} := \sum_{k} D(k,h_i) \]
\[ R^- := \max \{ N D_{\text{max}}, D_{\text{tot}} \times (1 + (N-1)/N) \} \quad \text{(Lower bound on response time)} \]
\[ R^+ := D_{\text{tot}} + (N-1) \times D_{\text{max}} \quad \text{(Upper bound on response time)} \]
\[ R := (R^+ + R^-)/2 \quad \text{(Compute average of upper and lower bounds)} \]

Return \( R \)